Why not SHA-3?

A glimpse at the heart of hash functions.

Alexis Breust, François Etcheverry

June 18, 2013

Abstract. October 2012, the NIST (National Institute of Standards and Technology) hash function competition ended selecting a new standard for hash algorithms: SHA-3. Based on a sponge construction, it is said to be one of the fastest and secured hash algorithms nowadays. This article points out that, indeed, some previous hash algorithms are not sufficient enough to assure security and states the strength of SHA-3.

Contents

1. Introduction: MD5

1.1. Cryptographic hash functions

Information security needs digital signatures which depend on the cryptographic strength of the underlying hash functions. In general, hash functions are used to improve security as well as efficiency. The work of a cryptographic hash function can be sum up with:

$$Message \longrightarrow \boxed{\begin{array}{c}\text{Cryptographic}\\ \text{hash function}\end{array}} \longrightarrow Digest$$

The digest is a lossy compression of a bit sequence (message) to a defined number of bits (512 is common nowadays). The digest is sometimes called (cryptographic) hash (value), checksum or (digital) fingerprint, even though these terms stand for functions with rather different properties and purposes. This paper will take the liberty to use the terms *hash function* instead of cryptographic hash function and *hash* instead of digest.

A message has a unique corresponding hash but different messages can have the same hash by the pigeonhole principle. However, this is what we want to avoid the most. To do so, as a minimum, a hash function must have the following properties:

- **Pre-image resistance.** Given a hash h it should be difficult to find any message m such that $h = \text{hash}(m)$.
- **Second pre-image resistance.** Given an input m_1 it should be difficult to find another input m_2 such that $m_1 \neq m_2$ and $\text{hash}(m_1) = \text{hash}(m_2)$.
- **Collision resistance.** It should be difficult to find two different messages m_1 and m_2 such that $\text{hash}(m_1) = \text{hash}(m_2)$. Such a pair is called a collision.

Estimating difficulty is quite a hard problem in computer science: appendix A tries to give some answers about the theoretical complexity. However, "difficult" is generally defined by "almost certainly beyond the reach of any adversary who must be prevented from breaking the system for as long as the security of the system is deemed important"[1].

1.2. MD4, you are not reliable any more

The MD4 (Message Digest 4) algorithm was developed by Ronald Rivest in 1990. The hash length is 128 bits. The algorithm has influenced later designs, such as the MD5, SHA-1 and RIPEMD algorithms.

The security of this algorithm has been often compromised. The first full collision attack against MD4 was published in 1995. In 2007, Sasaki *et al.* ([1], *"New Message Difference for MD4"*) can provide a collision in less than two MD4 hash operations. A theoretical pre-image attack also exists with a $O(2^{102})$ complexity ([2], 2008).

However, MD4 is still used for its speed of computation in hash tables, eMule P2P networks or more notably in NTLM password-derived key digests on Microsoft Windows[2] NT, XP, Vista and 7.

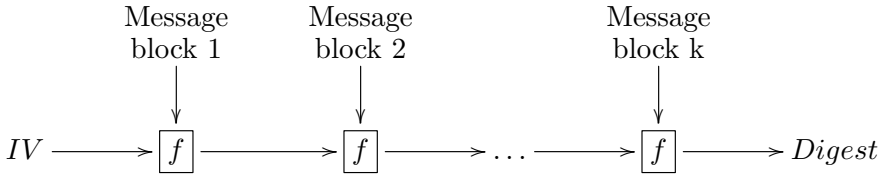
[1] `http://en.wikipedia.org/wiki/Cryptographic_hash_function`
[2] `http://msdn.microsoft.com/en-us/library/cc236715.aspx`

1.3. Definition of MD5

MD4's son, MD5, was created in 1991 in order to correct errors in its conception. MD5 algorithm produces a 128-bit hash value. Nevertheless, Wand and Yu have shown that MD5 is not collision resistant ([3], 2005). As such, MD5 is not suitable for applications like SSL certificates or digital signatures that rely on this property. Since its creation, and until a few years ago, MD5 was used, with SHA-1, as a norm for hash function. However, nowadays, MD5 and SHA-1 are still predominantly used in software and in the Internet.

Merkle-Damgård construction

The Merkle-Damgård construction, used in MD5 and SHA-1 for instance, can be summed up like this:



The first block is the *Initial Value* used to initialise the loop. The idea is to divide the original message into k blocks of m bits. If needed, the message can be padded by 1 and zeros to fix the last block length.

$$\begin{array}{rcccc} f & : & \{0,1\}^n \times \{0,1\}^m & \rightarrow & \{0,1\}^n \\ & & (C, M) & \mapsto & f(C, M) \end{array}$$

The resistance against collisions of the hash function is in part conditioning by the resistance of the f function. This function is what differs in MD4, MD5, SHA-1 or SHA-2.

However, the main problem of the Merkle-Damgård construction is the extension of a collision. Given two messages with the same hash $\text{hash}(m_1) = \text{hash}(m_2)$, all extensions will have the same hash: $\text{hash}(m_1||A) = \text{hash}(m_2||A)$ where $||$ denotes the concatenation. This particularity can be exploited by malicious developer to make *good* and *evil* programs with the same hash. More information in section 2.2.

MD5 algorithm overview

MD5 processes a message into a fixed-length output of 128 bits. The input message is broken up into chunks of 512-bit blocks. The main MD5 algorithm operates on a 128-bit state, divided into four 32-bit words, denoted a, b, c and d. These are initialized to certain fixed constants.

The following is devoted to describe the compression function for MD5. For each 512-bit block m_i of the padded message m, divide m_i into 32-bit words w_j. The compression algorithm for Mi has four rounds, and each round has 16 operations. Four successive step operations are as follows:

$$\begin{aligned} a &= b + ((a + \phi_i(b, c, d) + w_i + t_i) \ll s_i) \\ d &= a + ((d + \phi_{i+1}(a, b, c) + w_{i+1} + t_{i+1}) \ll s_{i+1}) \\ c &= d + ((c + \phi_{i+2}(d, a, b) + w_{i+2} + t_{i+2}) \ll s_{i+2}) \\ b &= c + ((b + \phi_{i+3}(c, d, a) + w_{i+3} + t_{i+3}) \ll s_{i+3}) \end{aligned}$$

where the operation + means addition modulo 2^{32}, t_{i+j} and s_{i+j} are step-dependent constants, w_{i+j} is a message word and $\ll$ is circularly left-shift.

Each round employs one non-linear round function given below :

$$\begin{aligned}
\phi_i(X,Y,Z) &= (X \wedge Y) \vee (\neg X \wedge Z) && 0 \leq i \leq 15 \\
\phi_i(X,Y,Z) &= (X \wedge Z) \vee (Y \wedge \neg Z) && 16 \leq i \leq 31 \\
\phi_i(X,Y,Z) &= X \oplus Y \oplus Z && 32 \leq i \leq 47 \\
\phi_i(X,Y,Z) &= Y \oplus (X \wedge \neg Z) && 48 \leq i \leq 63
\end{aligned}$$

where X, Y, Z are 32-bit words.

2. Weaknesses in MD5

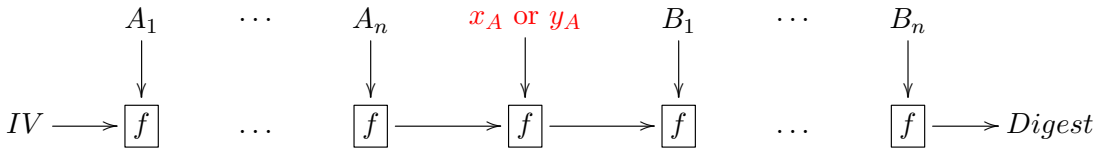
2.1. Differential attack on MD5

Weaknesses in MD5 have been found out by Hans Dobbertin in 1996 [4] and, since 2005, these weaknesses can be exploited by any-one, due to the work of Wang and Yu (see [3]). What are these weaknesses? The possibility to create collisions on demand, with chosen prefix. Wang and Yu claim that their attack can find a collision in about 2^{39} MD5 hash calculations, which would mean between 15 minutes and 1 hour on IBM P690. Actually, it took us less than half an hour to find a collision, using the Peter Selinger implementation of Wang and Yu's algorithm on 50 CPUs (using Ensibm, our dedicated server), without even being optimal with our parallelism (as the different programmes didn't share their data).

That way, one can easily find 512-bit MD5 collisions.

2.2. Attack on Merkle-Damgård

What can you do with such collisions? Say you found x and y 512-bit messages, such that $\text{MD5}(x) = \text{MD5}(y)$. As you can choose the prefix of your collision, you can take any message A with length a multiple of 512, and get $\text{MD5}(A||x_A) = \text{MD5}(A||y_A)$, where $||$ denotes concatenation. Be aware that you need that prefix before you compute a x and y. Then, you can use a propriety of the Merkle-Damgård construction to add any suffix to your collision, and that way, you can create two messages $A||x_A||B$ and $A||y_A||B$ with the same MD5 hash.



In other words, you can create a collision between two files, where only 512 bits are different, with the same MD5 digest. How can you exploit this? Well for example, you can create two executables `Prog1` and `Prog2`, with only 512 different bits, having the same hash and doing different things. How does it work? Imagine that your programmes are doing the following things:

- **Prog1**: `if (DATA1 == DATA1) do exec1 else do exec2;`
- **Prog1**: `if (DATA2 == DATA1) do exec1 else do exec2;`

As you might have understood it, your collision is DATA1/DATA2. You just have to modify exec1 and exec2 to get your programs to do whatever you want them to do. A proof that MD5 is really broken is that there is a lot of programmes circulating on the internet helping to use these vulnerability. For example, it took us only a few hour to find sources and methodology [3] to create our own programmes "good" and "evil" sharing the same MD5 hash.

2.3. Historical attacks

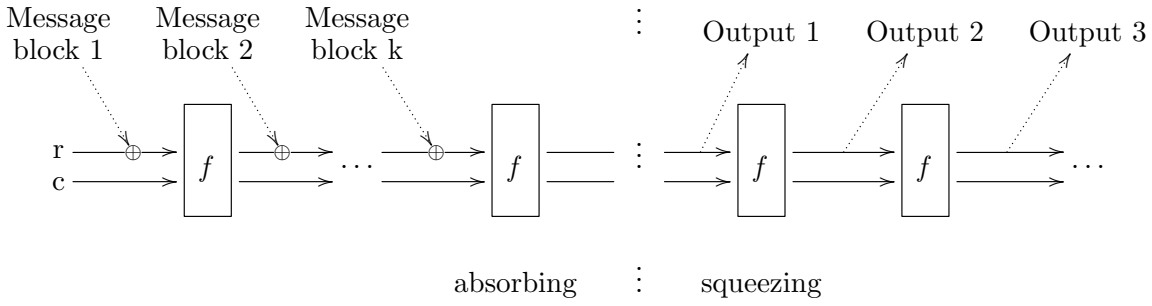
In September 2011 was discovered the virus Duqu, and later in May 2012 the virus FLAME. Both viruses use MD5 collisions *inter alia* to create false certificates. With the rest of the family (Stuxnet, Gauss), these malware are the bigger security threats at the moment. These viruses are one of the main reason for the organisation of the SHA-3 competition.

3. General attacks against SHA-3

3.1. Sponge function

Winner of the NIST[4] hash function competition in October 2012, Keccak submission ([5], 2011) uses a sponge function. Scarcely implemented in hash functions, the sponge function is a specificity of the newly declared standard SHA-3, a Keccak variant.

The sponge function, which is an adaptation from the RadioGatún structure ([6], 2009), can be quickly sketched as:



There are at least two notable differences between the sponge and the Merkle-Damgård construction. First, the sponge function can produce a hash with on demand length while Merkle-Damgård sets it. Then, the links between the compression functions are split into n and m bits.

$$\begin{array}{rcccc} f & : & \{0,1\}^r \times \{0,1\}^c & \rightarrow & \{0,1\}^r \times \{0,1\}^c \\ & & (r_i \oplus m_i, c_i) & \mapsto & (r_{i+1}, c_{i+1}) \end{array}$$

In Keccak[r,c] implementation, r is the *bitrate* and c the capacity. The sum $r + c$ determines the width of the Keccak-f permutation used in the sponge construction and is restricted to values in $\{25, 50, 100, 200, 400, 800, 1600\}$. Keccak allows one to choose its security parameter c independently from the output length.

In estimates of Keccak developers for the safety margin against different types of Keccak, it may be significant to notice that Keccak-f is supposed to have about twice as many rounds

[3] see `http://www.mathstat.dal.ca/~selinger/md5collision/`

[4] National Institute of Standards and Technology: `http://www.nist.gov/`

(24) as strictly required (13) for Keccak to stand up to its security claim, for any choice of the capacity.

Keccak algorithm seems quite resistant to attacks. Nowadays, there are only generalist ways, based on brute force methods, to break SHA-3. This paragraph is devoted to two of these attacks, trying to prove that practical results are very close to theoretical ones.

3.2. Pre-image attack

Given y a digest, how can we find an x such as $\mathrm{SHA3}(x) = y$? The most obvious method to do it is brute force. I choose to generate a lot of x and I hope that I will find one such as $\mathrm{SHA3}(x) = y$. Even if the method is quite simple, we must ask about its feasibility.

If our digest y is n bits long, hence there are 2^n possibility of digests. If Keccak is supposed to be ideal, every digest has a 2^{-n} chance to appear on a total random created x. So, one may need $O\left(2^n\right)$ tests to find out a valid x.

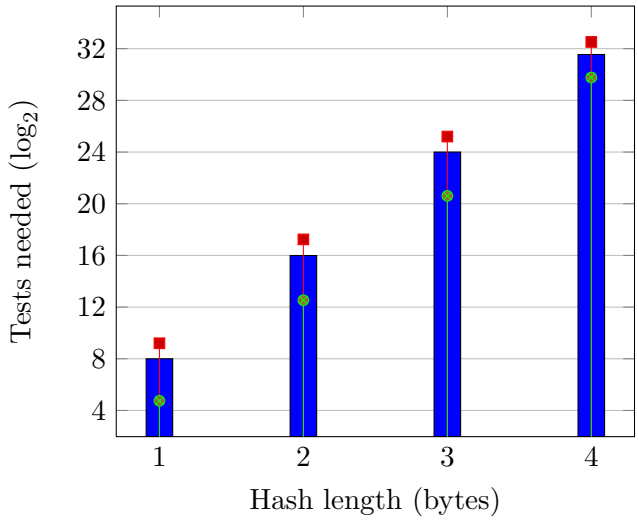


Figure 1: Average number of hashes ($\log_2$) needed to get a collision with a defined digest. First and ninth deciles are indicated.

Figure 1 presents the results of our test program. Given a Keccak digest, in our example the beginning of the 192-bit-long random digest `FD3E B0AB C323 513F C688 BFC5 8F55 1EDC 4F08 27D1 149D 58AA`, one wants to get a 16-byte-long message which gives the first n bits of the previous digest once hashed. Theory predicts that one may need 2^n random generated messages to get the correct beginning digest.

Results are averaged on 50000, 1000, 500 and 20 tests respectively for 1, 2, 3 and 4 bytes of the digest. Except for the latter, the number of tests is sufficient to state a good tendency in the statistics. The first three actually are under 0.1% of error, which confirms the theory. It is also interesting to notice that the variance of the results is also with $O(2^n)$ complexity.

3.3. Birthday attack

In probability theory, the birthday problem concerns the probability that, in a set of m randomly chosen people, some pair of them will have the same birthday. The cryptographic use of that problem is currently called the **birthday attack**. Given the Keccak hash function, we are considering $m = 2^n$ digests, where n is the number of bits of the digests.

The probability that, with x digests, all of them are completely different is simply calculated by

$$\bar{p}(x) = 1 \times \frac{m-1}{m} \times \frac{m-2}{m} \times \cdots \times \frac{m-x+1}{m}$$

Hence, the probability that at least two of them are the same is given by

$$p(x) = 1 - \frac{m!}{m^x (m-x)!}$$

An approximation can be found using $e^{-\frac{k}{m}} \approx 1 - \frac{k}{m}$ for each $k \ll m$ and gives

$$p(x) = 1 - e^{-\frac{x(x-1)}{2m}}$$

Hence

$$x(x-1) = -2m \log(1 - p(x))$$

Using $p(x) = 0.5$ (on average), we get (for $x > 0$):

$$\bar{x} = \frac{1}{2}\left(1 + \sqrt{8m\log(2) + 1}\right)$$

$$\boxed{\log_2(\bar{x}) \approx \frac{n+1}{2} + \frac{\log_2(\log(2))}{2}} \tag{1}$$

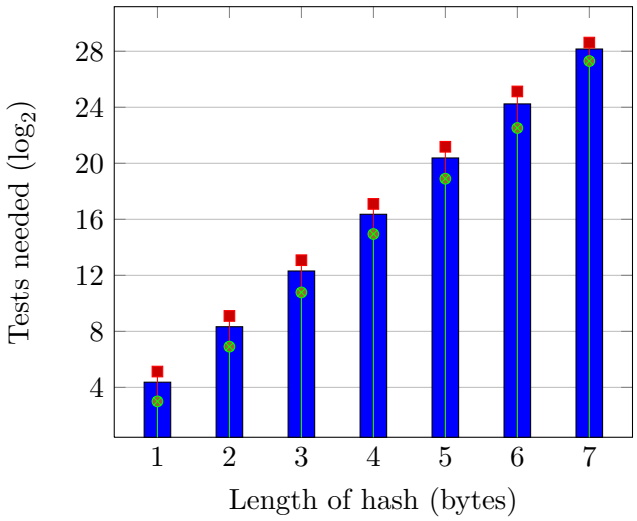


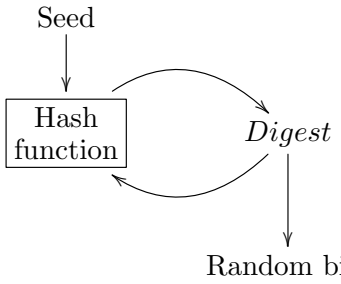
Figure 2: Average number of hashes ($\log_2$) needed to get a collision using the birthday attack. First and ninth deciles are indicated.

Figure 2 illustrates equation 1. To simplify, one may need $O\left(\sqrt{m}\right) = O\left(2^{n/2}\right)$ tests to find two messages with the same m-bit-long digest. This is a general attack that can be done on every hash function and proves that each one should be long enough. The reader should report to appendix A to see how big should be your digest to counter general attacks and to [7] which discuss the resistance of regular hash functions against birthday attacks.

Our test program generates successive random messages and checks if their digests (the first n bytes) have already been generated. Once two digests are identical, the program stops. We ran this program over 1000 times for each length of digest to get these average values.

However the results are almost what we expected for the birthday attack, all of them are distant for a very tiny value (0.1) from the theoretical prediction. Hence, we have to ask whether the hash function SHA-3 is sufficiently random or not. In order to do so, we implemented a iterative function which generates successive digests.

3.4. Randomness validity of SHA-3



Once seeded, the output of the function are stored into a file which is used to re-seed the function and so on. Now we can test the uniformity of the outputs of the function.

Our test creates one million self-iterating digests to build a 512000000-bit-long file. This file's entropy is tested using Federal Information Processing Standard (FIPS) 140-2 tests with the `rngtest` program. The reader may find more information in Caddy's *Encyclopedia of Cryptography and Security* ([8], 2011) about the FIPS.

The tests are done on blocks of 20000 bits, hence there are 26000 blocks per file. For each block, the FIPS 140-2 will tell whether the block has passed or not. On 50 files tested (with random initial seeds), there are exactly 22 on average failures over 26000. That is to say about 0.08% of failures.

In order to make a good comparison, we implemented the same protocol using `rand` from *libc* instead of SHA-3 digests. On average, we got 21.3 failures over 26000 blocks. This means SHA-3 can be used as a pseudo-random generator. However SHA-3 has to be architecture-optimized to be quick enough.

4. Differential attacks on SHA-3

4.1. Principle

Previous study prove that it is infeasible to lead a brute force attack against SHA-3, as soon as you chose a valid parameter. Yet, this is not a sufficient proof of security, as actually, a lot of broken hash functions are still not attackable via brute force. The main attack against hash functions is called the differential attack. What is it? The main idea is to study the hash function to find a set of conditions that two messages must verify in order to have a much better probability to collide.

How do you find such conditions? Actually, the main method is to study manually the function to find them. You study how a one-bit modification propagate in your function, and how you can try to stop this propagation with new bit modifications. Soon enough, you will find that, in order to perform these cancellations, you need to have some bits verifying conditions. These conditions have a certain probability to occur, which will decide the efficiency of your differential attack.

Actually, Keccak and its sponge function are too recent (or too efficient, who knows?) to have been attacked that way. In our study, we only tried to perform some tests to see if we could attack the randomness of SHA-3 by trying to work at bit-level.

4.2. Bit sensibility

A first step was to try finding a possibly less active bit. To do so, we used the following algorithm:

```
for i in 1 .. LOOP {
    generate random msg; // 1024 bit
    altered_msg = msg;
    for j in altered_msg {
        change bit j in altered_msg;
        compare (SHA3(msg), SHA3(altered_msg));
        change bit j in altered_msg;
    }
}
```

Where `compare` calculates the number of bits at different values between the two input message. The idea here is to calculate for each bit the average value (on loop) returned by `compare`, to see if any of the input bits has a more important role for SHA-3.

Here are the results for three different values of LOOP:

LOOP	min	max	average
10000	287.70	288.24	288.02
50000	287.80	288.18	287.99
100000	287.86	288.10	287.99

Table 1: SHA3 one-bit modification impact

288 being the limit if SHA-3 was a perfect hash function (one bit modification implying a completely new random results). We can see that there is no conclusion to collect here, and accordingly it is not relevant to use this data to find a less active bit.

For a second approach, we applied the same algorithm but using a weakened SHA-3 : as one of the most important method to equalize the importance between the bits in a hash function is to apply the same function several time (to help a modification to propagate itself in the system). We reduced the number of rounds in Keccak-f (in comparison, SHA-3 uses Keccak-f with 24 rounds). $LOOP = 100000$ in the next table.

rounds / bits	1-448	449-512	513-640	641-704	705-768	769-896	897-960	961-1024
1	287.86	119.37	116.56	124.12	119.88	121.86	9.50	7.00
2	288.00	288.00	288.00	288.00	288.00	288.00	119.37	116.75
3	288.00	288.00	288.00	288.00	288.00	288.00	287.78	288.21
4	288.00	288.00	288.00	288.00	288.00	288.00	288.00	288.00

We can see that the last bits of the input have a really small impact on the output of Keccak-f. In one round, only the first 448 bits have an acceptable impact on the output, and the bits 897 to 1024 have a really low impact which affects the output even after a second round. Yet, all that disappears after the fourth round.

The conclusion here is that even if some bits have a different impact on Keccak-f (from a random message), these differences are quickly erased by the followings rounds. We recall that Keccak developers considered 13 are enough rounds, and advised 24 rounds for SHA-3.

4.3. Bit correction

One of the principles with a differential attack is to try correcting a first modification using new modifications. Our idea here, was to minimize the number of modified bits, finding first a bit on first block which minimizes modifications of the digest, and then trying to cancel these modifications with a new one bit modification on a second block. As SHA-3 is too much resistant for us, we just tried to minimize the number of modifications on the final digest.

To do so, we applied the following method: for LOOP $2 * 1024$ random messages (so they are 2 SHA-3 block sized), find the 1-bit modification on the first block which minimizes the number of modified bits on the output. Then, keeping that modification, find the 1-bit modification on the second block which also minimizes the modification on the digest.

LOOP	absolute min	average min
5000	227	249.01
8000	220	249.08
10000	226	249.09

LOOP	absolute min	average min
5000	235	238.3
8000	207	238.3
10000	211	238.2

Table 2: SHA-3 1-bit correction on the left, $MIN_{1..1024}$(rand) on the right

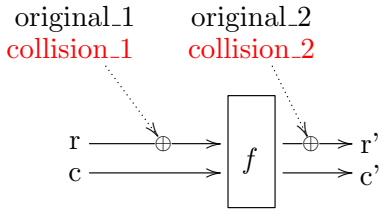
Having these results, the next question that came to us was : Are we getting something better than what we would obtain with a perfect hash function? Hence, we compared our results to the minimum number of bit at 1 in 1024 random 576-bit messages (as SHA3($original\_msg$) $\oplus$ SHA3($altered\_msg$) would be equivalent to such a random message if SHA-3 was perfect). We used `rand` from library *time.h* to generate each bit.

We came to two different conclusions, either `rand` has weaknesses, illustrated here, or a bit modification imply more modifications on SHA-3 digests than expected, which might be an interesting property to study in the future if proven. A theoretical study should be performed to have a definitive conclusion here.

Coming back on our first idea, as we didn't get expected results, we decided to perform the same test with a one-block sized message and only one modification. We obtain the same results (an average min at about 249) than with our one-bit correction.

Actually, this shows us that the method to generate differential attacks on a sponge construction will probably be different than a differential attack on a Merkle–Damgård construction, because of correcting a block with the next block is impossible here, due to the 2^c bits output that can not be modified.

This is the reason why finding an internal collision on SHA-3 is equivalent to finding a collision on these particular bits, which explain the theoretical security offered by SHA-3. This is due to the XOR construction.



A little proof for this equivalence: Given a collision on c', you can create a collision on $r'||c'$ using $collision\_2 = original\_2 \oplus r\_output\_errors$. Reciprocally, if you found a collision on the output $r'||c'$, then you have a collision on c'.

This research suggest that future differential attacks on SHA-3 will probably be collisions on one block, or at least, second block modifications will be trivial.

4.4. Conclusion

Keccak certainly differs from all functions of the Merkle-Damgård family (MD5, SHA-1, SHA-2) as it uses a sponge function construction. This fact can be considered neither as an advantage nor a disadvantage. Sponge functions are new constructions that will now, thanks to the nomination of Keccak as SHA-3, attract more attention from cryptographers. Certainly this is a minus as this construction is not very well analysed yet.

Our study states the strength of Keccak as it does not get easily trapped into simple differential attacks. The number of rounds setted by the developers seams sufficient to avoid these simple attacks. Moreover, the security parameter c, which cannot be directly affected by the input, provides a good reason for considering gladly sponge functions.

It is a positive fact that SHA-3 has not the same structure as SHA-2. That means that a new attack could not threaten both algorithms at the same time. Moreover, as SHA-2 is still very solid and is not going to be replaced by Keccak soon.

Appendices

A. How big should be 2^n?

The Keccak web site[5] gives hints to evaluate the magnitude of 2^n numbers. Here are summed up some of them in order to give the idea of a sufficient security.

Not enough: 2^{64}

Imagine an attacker who want to crack your 128-bit-long digest by birthday attack. If he had access to one billion computers, each performing one billion evaluation of your hash function per second, it would take **18.4 seconds** to evaluate the permutation 2^{64} times.

Sufficient for today: 2^{128}

The same attack on a hash of 256 bits would take $\mathbf{1.1 \times 10^{13}}$ **years** (770 times the estimated age of the universe) to evaluate the permutation 2^{128} times.

Surely good: 2^{256}

The same attack on a hash of 512 bits would take $\mathbf{3.7 \times 10^{51}}$ **years** (2.6×10^{41} times the estimated age of the universe) to evaluate the permutation 2^{256} times.

Considering an irreversible computer working at 2.735°K (the average temperature of the universe), Landauer's principle implies that it cannot consume less than 2.62×10^{-23} joule every time a bit is changed. (Computers actually consume much more than that.) Just counting from 1 to 2^{256} would take at least 3×10^{54} joules (the total energy output of the Sun during 2.5×10^{20} years).

Conclusion

A simple way to evaluate security is to decide arbitrary that nowadays $O(2^{100})$ begins to be a sufficient complexity. In the future, if computation increases a lot more, it would be necessary to extend minimal hash length.

[5] `http://keccak.noekeon.org/tune.html`

References

[1] Y. Sasaki, L. Wang, K. Ohta, and N. Kunihiro, "New message difference for md4," in *Fast Software Encryption, 14th International Workshop, FSE 2007, Luxembourg, Luxembourg, March 26-28, 2007, Revised Selected Papers* (A. Biryukov, ed.), vol. 4593 of *Lecture Notes in Computer Science*, pp. 329–348, Springer, 2007.

[2] G. Leurent, "Md4 is not one-way," in *FSE*, pp. 412–428, 2008.

[3] X. Wang and H. Yu, "How to break md5 and other hash functions," in *In EUROCRYPT*, Springer-Verlag, 2005.

[4] H. Dobbertin, "Cryptanalysis of md5 compress," May 1996.

[5] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "The Keccak SHA-3 submission," January 2011. http://keccak.noekeon.org/.

[6] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "The road from panama to keccak via radiogatún," in *Symmetric Cryptography* (H. Handschuh, S. Lucks, B. Preneel, and P. Rogaway, eds.), no. 09031 in Dagstuhl Seminar Proceedings, (Dagstuhl, Germany), Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009.

[7] N. Mouha, G. Sekar, and B. Preneel, "Challenging the increased resistance of regular hash functions against birthday attacks.," 2012.

[8] T. Caddy, "Fips 140-2.," in *Encyclopedia of Cryptography and Security (2nd Ed.)*, pp. 468–471, 2011.

[9] K. Pearson, "On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that can be reasonably supposed to have arisen from random sampling," *Philosophical Magazine*, vol. 50, pp. 157–175, 1900.

[10] T. Nipkow and C. Pusch, "Avl trees.," 2004.

[11] S. Knellwolf and D. Khovratovich, "New preimage attacks against reduced sha-1.," p. 440, 2012.